Οι ακόλουθες σημειώσεις έχουν προέλθει από το Concurrency Java Tutorial της Oracle (https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)

Ταυτοχρονισμός (concurrency)

Διεργασίες και νήματα (processes and threads)

Διεργασίες

Μια διεργασία διαθέτει ένα αυτοδύναμο περιβάλλον εκτέλεσης. Κάθε διεργασία έχει το δικό της ιδιωτικό σύνολο από πόρους που χρησιμοποιεί κατά την εκτέλεσή της όπως ένα τμήμα της μνήμης που μπορεί να χρησιμοποιήσει μόνο η ίδια. Η επικοινωνία με τις άλλες διεργασίες γίνεται μέσω μηχανισμών IPC (Inter Process Communication) όπως είναι τα pipes και τα sockets. Μια εφαρμογή σε Java μπορεί να δημιουργήσει επιπλέον διεργασίες με το αντικείμενο ProcessBuilder.

Νήματα

Τα νήματα υφίστανται εντός μιας διεργασίας και μοιράζονται τους πόρους που διαθέτει η διεργασία όπως είναι η μνήμη και αρχεία της διεργασίας τα οποία είναι ανοικτά. Κάθε πρόγραμμα Java ξεκινά με ένα νήμα που ονομάζεται main thread και έχει τη δυνατότητα να δημιουργήσει νέα νήματα.

Αντικείμενα Νημάτων

Κάθε νήμα σχετίζεται με ένα στιγμιότυπο της κλάσης Thread. Για να δημιουργηθεί μια εφαρμογή που να υποστηρίζει ταυτοχρονισμό μπορεί είτε να δημιουργούνται απευθείας τα νήματα που εκτελούν ασύγχρονα τις επιμέρους εργασίες είτε να χρησιμοποιηθεί ένας executor για τη διαχείριση των επιμέρους εργασιών.

Ορισμός και εκκίνηση ενός νήματος

Ένα νήμα μπορεί να δημιουργηθεί είτε παρέχοντας ένα αντικείμενο που υλοποιεί το Runnable interface είτε δημιουργώντας μια υποκλάση της Thread.

```
public class HelloRunnable implements Runnable {

  public void run() { System.out.println("Hello from a thread!"); }

  public static void main(String args[]) {
    (new Thread(new HelloRunnable())).start();
  }
}
```

HelloRunnable.java

```
javac HelloRunnable.java
java HelloRunnable
```

Hello from a thread!

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }
```

```
    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

HelloThread.java

```
javac HelloThread.java
java HelloThread
```

Hello from a thread!

Παύση εκτέλεσης με sleep

Η Thread.sleep αναστέλλει την εκτέλεση του τρέχοντος νήματος για μια συγκεκριμένη χρονική περίοδο.

```
public class SleepMessages {
  public static void main(String args[]) throws InterruptedException {
    String importantInfo[] = {"Mares eat oats", "Does eat oats",
                              "Little lambs eat ivy", "A kid will eat ivy too"};

    for (int i = 0; i < importantInfo.length; i++) {
      // Pause for 4 seconds
      Thread.sleep(4000);
      // Print a message
      System.out.println(importantInfo[i]);
    }
  }
}
```

SleepMessages.java

```
javac SleepMessages.java
java SleepMessages
```

```
Mares eat oats
Does eat oats
Little lambs eat ivy
A kid will eat ivy too
```

Interrupts

Ένα interrupt είναι ένα σήμα προς ένα νήμα ότι θα πρέπει να διακόψει ότι έκανε και να κάνει κάτι άλλο. Ο προγραμματιστής καθορίζει τι θα κάνει το νήμα όταν δεχθεί το interrupt αλλά συχνά αυτό που συμβαίνει είναι ότι το νήμα τερματίζεται. Ένα νήμα στέλνει διακοπή καλώντας τη μέθοδο interrupt στο αντικείμενο thread που πρόκειται να διακοπεί. Για να λειτουργήσει σωστά η διακοπή θα πρέπει το νήμα να υποστηρίζει τη διακοπή του.

Joins

Η μέθοδος join επιτρέπει σε ένα νήμα να περιμένει την ολοκλήρωση της εκτέλεσης ενός άλλου νήματος.

Το παράδειγμα SimpleThreads

Ένα νήμα της κλάσης MessageLoop εμφανίζει κάθε 4 δευτερόλεπτα ένα μήνυμα μέχρι να εμφανιστούν τα 4 διαθέσιμα μηνύματα. Ανάλογα με το όρισμα γραμμής εντολών κατά την εκτέλεση το main thread θα περιμένει

```
public class SimpleThreads {
```

```
// Display a message, preceded by
// the name of the current thread
static void threadMessage(String message) {
  String threadName = Thread.currentThread().getName();
  System.out.format("%s: %s%n", threadName, message);
}

private static class MessageLoop implements Runnable {
  public void run() {
    String importantInfo[] = {"Mares eat oats", "Does eat oats",
                              "Little lambs eat ivy",
                              "A kid will eat ivy too"};
    try {
      for (int i = 0; i < importantInfo.length; i++) {
        // Pause for 4 seconds
        Thread.sleep(4000);
        // Print a message
        threadMessage(importantInfo[i]);
      }
    } catch (InterruptedException e) {
      threadMessage("I wasn't done!");
    }
  }
}

public static void main(String args[]) throws InterruptedException {

  // Delay, in milliseconds before
  // we interrupt MessageLoop
  // thread (default one hour).
  long patience = 1000 * 60 * 60;

  // If command line argument
  // present, gives patience
  // in seconds.
  if (args.length > 0) {
    try {
      patience = Long.parseLong(args[0]) * 1000;
    } catch (NumberFormatException e) {
      System.err.println("Argument must be an integer.");
      System.exit(1);
    }
  }

  threadMessage("Starting MessageLoop thread");
  long startTime = System.currentTimeMillis();
  Thread t = new Thread(new MessageLoop());
  t.start();

  threadMessage("Waiting for MessageLoop thread to finish");
  // loop until MessageLoop
```

```
    // thread exits
    while (t.isAlive()) {
      threadMessage("Still waiting...");
      // Wait maximum of 1 second
      // for MessageLoop thread
      // to finish.
      t.join(1000);
      if (((System.currentTimeMillis() - startTime) > patience) &&
          t.isAlive()) {
        threadMessage("Tired of waiting!");
        t.interrupt();
        // Shouldn't be long now
        // -- wait indefinitely
        t.join();
      }
    }
    threadMessage("Finally!");
  }
}
```

SimpleThreads.java

```
javac SimpleThreads.java
java SimpleThreads
```

```
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Little lambs eat ivy
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: A kid will eat ivy too
main: Finally!
```

```
java SimpleThreads 10
```

```
main: Starting MessageLoop
main: Waiting for MessageLo
main: Still waiting...
main: Still waiting...
```

```
main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
main: Still waiting...
main: Still waiting...
main: Tired of waiting!
Thread-0: I wasn't done!
main: Finally!
```

Συγχρονισμός

Τα νήματα επικοινωνούν μέσω διαμοιρασμού πεδίων. Αν και αυτός ο τρόπος επικοινωνίας είναι αποδοτικός μπορεί να παρουσιαστούν προβλήματα παρεμβολής νημάτων και λαθών συνέπειας μνήμης. Τα προβλήματα αυτά αντιμετωπίζονται με συγχρονισμό.

Παρεμβολή νημάτων (thread interference)

```
class Counter {
  private int c = 0;

  public void increment() { c++; }

  public void decrement() { c--; }

  public int value() { return c; }
}
```

Counter.java

Η εντολή c++ αναλύεται σε 3 επιμέρους βήματα.

1. Ανάκληση της τρέχουσας τιμής του c
2. Αύξηση της ανακληθείσας τιμής κατά 1
3. Αποθήκευση της αυξημένης τιμής πίσω στο c

Το ίδιο ισχύει και για την εντολή c--.

Τα βήματα εκτέλεσης της εντολής c++ ενός νήματος μπορεί να παρεμβληθούν στα βήματα εκτέλεσης ενός άλλου νήματος που εκτελεί την ίδια περίπου χρονική στιγμή την εντολή c-- και να λάβουμε διαφορετικό αποτέλεσμα από το αναμενόμενο όπως στο ακόλουθο παράδειγμα στο οποίο το threadA εκτελεί τη μέθοδο increment ενώ το threadB εκτελεί τη μέθοδο decrement:

Thread A: Ανάκληση του c.
Thread B: Ανάκληση του c.
Thread A: Αύξηση της ανακληθείσας τιμής, το αποτέλεσμα είναι 1.
Thread B: Μείωση της ανακληθείσας τιμής; Το αποτέλεσμα είναι -1.
Thread A: Αποθήκευση αποτελέσματος στο c; το c είναι τώρα 1.
Thread B: Αποθήκευση αποτελέσματος στο c; το c είναι τώρα -1.

```
class CounterExample {
  static Counter c = new Counter();
```

```
  private static class IncrementCounter implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++) {
        if (i % 2 == 0)
          c.increment();
        else
          c.decrement();
      }
    }
  }

  public static void main(String[] args)  throws InterruptedException {
    Thread[] threads = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads[i] = new Thread(new IncrementCounter());

    System.out.println(c.value());

    for (int i = 0; i < 100; i++)
      threads[i].start();

    for (int i = 0; i < 100; i++)
      threads[i].join();

    System.out.println(c.value());
  }
}
```

CounterExample.java

```
javac CounterExample.java
java CounterExample
```

```
0
-3
```

Λάθη συνέπειας μνήμης

Συγχρονισμένες μέθοδοι

Μια μέθοδος γίνεται συγχρονισμένη αν προστεθεί η λέξη synchronized στον ορισμό της. Δεν επιτρέπεται δύο κλήσεις μιας synchronized μεθόδου να παρεμβληθούν μεταξύ τους. Όταν ολοκληρώσει την εκτέλεσή της μια συγχρονισμένη μέθοδος τότε είναι εγγυημένο ότι όλες οι αλλαγές που έχει κάνει στην κατάσταση του αντικειμένου είναι ορατές από τα άλλα νήματα.

```
class SynchronizedCounter {
  private int c = 0;

  public synchronized void  increment() { c++; }

  public synchronized void  decrement() { c--; }

  public synchronized int value() { return c; }
```

```
}
```
SynchronizedCounter.java

```
class SynchronizedCounterExample {
  static SynchronizedCounter c = new SynchronizedCounter();
  private static class IncrementCounter implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++) {
        if (i % 2 == 0)
          c.increment();
        else
          c.decrement();
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread[] threads = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads[i] = new Thread(new IncrementCounter());

    System.out.println(c.value());

    for (int i = 0; i < 100; i++)
      threads[i].start();

    for (int i = 0; i < 100; i++)
      threads[i].join();

    System.out.println(c.value());
  }
}
```
SynchronizedCounterExample.java

```
javac SynchronizedCounterExample.java
java SynchronizedCounterExample
```
```
0
0
```

Εγγενές κλείδωμα (intrinsic lock) και συγχρονισμός

Κάθε αντικείμενο έχει ένα intrinsic lock συνδεδεμένο με αυτό. Αν ένα νήμα έχει το intrinsic lock ενός αντικειμένου κανένα άλλο νήμα δεν μπορεί να αποκτήσει το ίδιο lock.

Όταν ένα νήμα καλεί μια synchronized μέθοδο τότε αυτόματα δεσμεύει το intrinsic lock του αντικειμένου της μεθόδου και το αποδεσμεύει όταν η μέθοδος επιστρέφει.

```
class SynchronizedCounter2 {
  private int c = 0;

  public void increment() {
    synchronized (this) { c++; }
```

```
  }

  public void decrement() {
    synchronized (this) { c--; }
  }

  public int value() {
    synchronized (this) { return c; }
  }
}
```

SynchronizedCounter2.java

```
class SynchronizedCounter2Example {
  static SynchronizedCounter2 c = new SynchronizedCounter2();
  private static class IncrementCounter implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++) {
        if (i % 2 == 0)
          c.increment();
        else
          c.decrement();
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread[] threads = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads[i] = new Thread(new IncrementCounter());

    System.out.println(c.value());

    for (int i = 0; i < 100; i++)
      threads[i].start();

    for (int i = 0; i < 100; i++)
      threads[i].join();

    System.out.println(c.value());
  }
}
```

SynchronizedCounter2Example.java

```
javac SynchronizedCounter2Example.java
java SynchronizedCounter2Example
```

```
0
0
```

```
public class MsLunch {
    private long c1 = 0;
```

```
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
            // c2++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
            // c1++;
        }
    }

    public void print(){
        System.out.printf("c1=%d c2=%d\n", c1, c2);
    }
}
```

MsLunch.java

```
public class MsLunchExample {
  static MsLunch c = new MsLunch();
  private static class IncrementCounter1 implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++)
        c.inc1();
    }
  }

  private static class IncrementCounter2 implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++)
        c.inc2();
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread threads1[] = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads1[i] = new Thread(new IncrementCounter1());

    Thread threads2[] = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads2[i] = new Thread(new IncrementCounter2());

    for (int i = 0; i < 100; i++) {
      threads1[i].start();
```

```
      threads2[i].start();
    }

    for (int i = 0; i < 100; i++) {
      threads1[i].join();
      threads2[i].join();
    }
    c.print();
  }
}
```

MsLunchExample.java

```
javac MsLunchExample.java
java MsLunchExample
```

c1=10000 c2=10000

Liveness

Η ικανότητα μιας εφαρμογής να μην μπλοκάρεται κατά την εκτέλεσή της ονομάζεται liveness.

Αδιέξοδο (deadlock)

Το αδιέξοδο περιγράφει μια κατάσταση στην οποία δύο ή περισσότερα νήματα μπλοκάρονται επ' αόριστο περιμένοντας το ένα το άλλο. Έστω το εξής παράδειγμα:

Ο Alphonse και ο Gaston είναι δύο ευγενείς που ακολουθούν τον κανόνα ότι όταν υποκλίνονται σε κάποιον θα πρέπει να παραμένουν στη θέση υπόκλισης μέχρι η υπόκλιση να ανταποδοθεί.

```
public class Deadlock {
  static class Friend {
    private final String name;
    public Friend(String name) { this.name = name; }
    public String getName() { return this.name; }
    public synchronized void bow(Friend bower) {
      System.out.format("%s: %s"
                        + "  has bowed to me!%n",
                        this.name, bower.getName());
      bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
      System.out.format("%s: %s"
                        + " has bowed back to me!%n",
                        this.name, bower.getName());
    }
  }

  public static void main(String[] args) {
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
      public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
```

```
      public void run() { gaston.bow(alphonse); }
    }).start();
  }
}
```

Deadlock.java

```
javac Deadlock.java
java Deadlock
```

```
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed to me!

Πέφτει σε deadlock
```

http://stackoverflow.com/questions/24718676/gaston-and-alphonse-example-how-does-the-bowback-get-accessed

Λιμοκτονία (starvation)

Η λιμοκτονία αφορά μια κατάσταση στην οποία ένα νήμα δεν μπορεί να προχωρήσει στην εκτέλεση της εργασίας του καθώς απαιτούμενοι πόροι για αυτή δεν είναι διαθέσιμοι λόγω του ότι άλλα νήματα τους παρακρατούν για μεγάλες περιόδους.

Livelock

Ένα νήμα συχνά ενεργεί ως απόκριση στις ενέργειες ενός άλλου νήματος. Αν το άλλο νήμα επίσης ενεργεί ως απόκριση στις ενέργειες ενός άλλου νήματος τότε μπορεί να συμβεί livelock. Όπως και στην περίπτωση του deadlock τα νήματα δεν μπορούν να σημειώσουν πρόοδο αλλά αντί να είναι μπλοκαρισμένα το ένα ανταποκρίνεται σε ενέργειες του άλλου χωρίς ουσιαστικά να παρατηρείται πρόοδος.

Guarded Blocks

Συχνά τα νήματα πρέπει να συντονίζουν τις ενέργειές τους. Ένας κοινός μηχανισμός συντονισμού είναι τα guarded blocks στα οποία ελέγχεται μια συνθήκη η οποία θα πρέπει να είναι αληθής για να προχωρήσει η εκτέλεση του block.

```
public class Drop {
  // Message sent from producer
  // to consumer.
  private String message;
  // True if consumer should wait
  // for producer to send message,
  // false if producer should wait for
  // consumer to retrieve message.
  private boolean empty = true;

  public synchronized String take() {
    // Wait until message is
    // available.
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    // Toggle status.
    empty = true;
    // Notify producer that
```

```
    // status has changed.
    notifyAll();
    return message;
  }

  public synchronized void put(String message) {
    // Wait until message has
    // been retrieved.
    while (!empty) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    // Toggle status.
    empty = false;
    // Store message.
    this.message = message;
    // Notify consumer that status
    // has changed.
    notifyAll();
  }
}
```

Drop.java

```
import java.util.Random;

public class Producer implements Runnable {
  private Drop drop;

  public Producer(Drop drop) { this.drop = drop; }

  public void run() {
    String importantInfo[] = {"Mares eat oats", "Does eat oats",
                              "Little lambs eat ivy", "A kid will eat ivy too"};
    Random random = new Random();

    for (int i = 0; i < importantInfo.length; i++) {
      drop.put(importantInfo[i]);
      try {
        Thread.sleep(random.nextInt(5000));
      } catch (InterruptedException e) {
      }
    }
    drop.put("DONE");
  }
}
```

Producer.java

```
import java.util.Random;
```

```
public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

Consumer.java

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

ProducerConsumerExample.java

```
javac ProducerConsumerExample.java
java ProducerConsumerExample
```

```
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```

Ο ενδεδειγμένος τρόπος δημιουργίας ενός σεναρίου παραγωγού καταναλωτή είναι χρησιμοποιώντας ένα BlockingQueue.

```
import java.util.Random;
import java.util.concurrent.BlockingQueue;

public class Producer2 implements Runnable {
    private BlockingQueue<String> drop;

    public Producer2(BlockingQueue<String> drop) {
        this.drop = drop;
    }
```

```
    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        try {
            for (int i = 0;
                 i < importantInfo.length;
                 i++) {
                drop.put(importantInfo[i]);
                Thread.sleep(random.nextInt(5000));
            }
            drop.put("DONE");
        } catch (InterruptedException e) {}
    }
}
```

Producer2.java

```
import java.util.Random;
import java.util.concurrent.BlockingQueue;

public class Consumer2 implements Runnable {
    private BlockingQueue<String> drop;

    public Consumer2(BlockingQueue<String> drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        try {
            for (String message = drop.take();
                 ! message.equals("DONE");
                 message = drop.take()) {
                System.out.format("MESSAGE RECEIVED: %s%n",
                                  message);
                Thread.sleep(random.nextInt(5000));
            }
        } catch (InterruptedException e) {}
    }
}
```

Consumer2.java

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.SynchronousQueue;
```

```
public class ProducerConsumerBlockingQueueExample {
    public static void main(String[] args) {
        BlockingQueue<String> drop =
            new SynchronousQueue<String> ();
        (new Thread(new Producer2(drop))).start();
        (new Thread(new Consumer2(drop))).start();
    }
}
```

ProducerConsumerBlockingQueueExample.java

```
javac ProducerConsumerBlockingQueueExample.java
java ProducerConsumerBlockingQueueExample
```

```
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```

Immutable objects

Ένα αντικείμενο θεωρείται immutable όταν η κατάστασή του δεν μπορεί να αλλάξει μετά τη δημιουργία του. Η χρήση immutable αντικειμένων θεωρείται ως καλή στρατηγική για τη δημιουργία απλού και αξιόπιστου κώδικα.

Ένα παράδειγμα Synchronized Class

```
public class SynchronizedRGB {

  // Values must be between 0 and 255.
  private int red;
  private int green;
  private int blue;
  private String name;

  private void check(int red, int green, int blue) {
    if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 ||
        blue > 255) {
      throw new IllegalArgumentException();
    }
  }

  public SynchronizedRGB(int red, int green, int blue, String name) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.name = name;
  }

  public void set(int red, int green, int blue, String name) {
    check(red, green, blue);
    synchronized (this) {
      this.red = red;
      this.green = green;
```

```
      this.blue = blue;
      this.name = name;
    }
  }

  public synchronized int getRGB() {
    return ((red << 16) | (green << 8) | blue);
  }

  public synchronized String getName() { return name; }

  public synchronized void invert() {
    red = 255 - red;
    green = 255 - green;
    blue = 255 - blue;
    name = "Inverse of " + name;
  }
}
```

SynchronizedRGB.java

Μια στρατηγική για ορισμό immutable αντικειμένων

1. Όχι "setter" μέθοδοι.
2. Όλα τα πεδία να είναι final και private.
3. Να μην επιτρέπεται σε υποκλάσεις να κάνουν override μεθόδους (π.χ. ορίζοντας την κλάση ως final).
4. Αν τα πεδία του στιγμιότυπου έχουν αναφορές σε mutable αντικείμενα, να μην επιτρέπεται σε αυτά τα αντικείμενα να αλλάξουν:
    a. Να μην παρέχονται μέθοδοι που τροποποιούν τα mutable αντικείμενα.
    b. Να μην διαμοιράζονται αναφορές σε mutable αντικείμενα. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

```
final public class ImmutableRGB {

  // Values must be between 0 and 255.
  final private int red;
  final private int green;
  final private int blue;
  final private String name;

  private void check(int red, int green, int blue) {
    if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 ||
        blue > 255) {
      throw new IllegalArgumentException();
    }
  }

  public ImmutableRGB(int red, int green, int blue, String name) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.name = name;
```

```
  }

  public int getRGB() { return ((red << 16) | (green << 8) | blue); }

  public String getName() { return name; }

  public ImmutableRGB invert() {
    return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
                            "Inverse of " + name);
  }
}
```

ImmutableRGB.java

Αντικείμενα concurrency υψηλού επιπέδου

Lock objects

Μόνο ένα νήμα μπορεί να έχει στην ιδιοκτησία του ένα Lock αντικείμενο σε κάθε χρονική στιγμή. Τα Lock αντικείμενα υποστηρίζουν το μηχανισμό wait/notify μέσω σχετιζόμενων Condition αντικειμένων.

Το κύριο πλεονέκτημα των Lock αντικειμένων σε σχέση με τα εγγενή locks είναι η δυνατότητα υποχώρησης. Η μέθοδος tryLock υποχωρεί αν το lock δεν είναι άμεσα διαθέσιμο ή αν δεν είναι διαθέσιμο πριν από κάποιο χρονικό διάστημα.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
  static class Friend {
    private final String name;
    private final Lock lock = new ReentrantLock();

    public Friend(String name) { this.name = name; }

    public String getName() { return this.name; }

    public boolean impendingBow(Friend bower) {
      Boolean myLock = false;
      Boolean yourLock = false;
      try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
      } finally {
        if (!(myLock && yourLock)) {
          if (myLock) {
            lock.unlock();
          }
          if (yourLock) {
            bower.lock.unlock();
          }
        }
      }
```

```
      return myLock && yourLock;
    }

    public void bow(Friend bower) {
      if (impendingBow(bower)) {
        try {
          System.out.format("%s: %s has"
                                + " bowed to me!%n",
                            this.name, bower.getName());
          bower.bowBack(this);
        } finally {
          lock.unlock();
          bower.lock.unlock();
        }
      } else {
        System.out.format("%s: %s started"
                              + " to bow to me, but saw that"
                              + " I was already bowing to"
                              + " him.%n",
                          this.name, bower.getName());
      }
    }

    public void bowBack(Friend bower) {
      System.out.format("%s: %s has"
                            + " bowed back to me!%n",
                        this.name, bower.getName());
    }
  }

  static class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;

    public BowLoop(Friend bower, Friend bowee) {
      this.bower = bower;
      this.bowee = bowee;
    }

    public void run() {
      Random random = new Random();
      for (;;) {
        try {
          Thread.sleep(random.nextInt(10));
        } catch (InterruptedException e) {
        }
        bowee.bow(bower);
      }
    }
  }

  public static void main(String[] args) {
```

```
    final Friend alphonse = new Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new BowLoop(alphonse, gaston)).start();
    new Thread(new BowLoop(gaston, alphonse)).start();
  }
}
```

Safelock.java

```
javac Safelock.java
java Safelock
```

```
Alphonse: Gaston has bowed to me!
Gaston: Alphonse started to bow to me, but saw that I was already bowing to him.
Gaston: Alphonse has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
Gaston: Alphonse has bowed to me!
Alphonse: Gaston has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
Gaston: Alphonse has bowed to me!
Alphonse: Gaston has bowed back to me!
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed back to me!
…
```

Executors

Executor Interfaces

Το package java.util.concurrent ορίζει 3 executor interfaces:

1. Executor: απλό interface που υποστηρίζει την εκκίνηση νέων εργασιών.
2. ExecutorService: subinterface του Executor, που προσθέτει δυνατότητες διαχείρισης του κύκλου ζωής τόσο των επιμέρους εργασιών όσο και του ίδιου του executor.
3. ScheduledExecutorService: subinterface του ExecutorService, που υποστηρίζει μελλοντική και περιοδική εκτέλεση εργασιών.

Executor interface

Αν r είναι ένα Runnable αντικείμενο και e είναι ένα αντικείμενο Executor η εκκίνηση του νήματος γίνεται με την εντολή: e.execute(r);

ExecutorService interface

Προσθήκη της μεθόδου submit η οποία δέχεται Runnable αλλά και Callable αντικείμενα και επιστρέφει ένα Future αντικείμενο

ScheduledExecutorService

Προσθήκη της μεθόδου schedule η οποία εκτελεί Runnable ή Callable εργασίες μετά από μια συγκεκριμένη καθυστέρηση.

Thread Pools

Τα thread pools αποτελούνται από worker threads τα οποία ελαχιστοποιούν την επιβάρυνση της δημιουργίας νέων νημάτων.

Ένας κοινός τύπος thread pool είναι το fixed thread pool το οποίο έχει πάντα έναν αριθμό από threads σε εκτέλεση. Αν ένα νήμα τερματιστεί τότε αυτόματα αντικαθίσταται από ένα νέο thread. Οι εργασίες υποβάλλονται στο thread pool μέσω μιας εσωτερικής ουράς η οποία διατηρεί όσες εργασίες δεν μπορούν να εξυπηρετηθούν λόγω του ότι οι είναι περισσότερες από τα διαθέσιμα νήματα.

Ένας τρόπος να δημιουργηθεί ένας executor ο οποίος χρησιμοποιεί ένα fixed thread pool είναι να κληθεί η new FixedThreadPool στο java.util.concurrent.Executors.

Fork/Join (JDK 7)

Το fork/join framework αποτελεί μια υλοποίηση του interface ExecutorService που εκμεταλλεύεται τους πολλαπλούς πυρήνες με τους οποίους μπορεί να είναι εφοδιασμένο ένα υπολογιστικό σύστημα. Είναι σχεδιασμένο για εργασίες οι οποίες μπορούν να διασπαστούν σε μικρότερα τμήματα αναδρομικά και ο στόχος είναι η χρήση όλης της επεξεργαστικής ισχύος έτσι ώστε να ενισχυθεί η απόδοση της εφαρμογής.

Concurrent collections

Το πακέτο java.util.concurrent προσθέτει ορισμένα collection interfaces στο Java Collections Framework όπως είναι το BlockingQueue και το ConcurrentMap.

Το BlockingQueue ορίζει μια δομή first-in-first-out η οποία μπλοκάρει όταν γίνεται απόπειρα να προστεθεί ένα στοιχείο σε μια ήδη γεμάτη ουρά ή όταν γίνεται προσπάθεια ανάκλησης ενός στοιχείου από μια άδεια ουρά.

Το ConcurrentMap είναι ένα subinterface του java.util.Map που ορίζει ατομικές λειτουργίες. Υλοποίηση ενός ConcurrentMap είναι το ConcurrentHashMap.

Atomic variables

Το πακέτο java.util.concurrent.atomic ορίζει κλάσεις που υποστηρίζουν ατομικές λειτουργίες σε απλές μεταβλητές. Οι κλάσεις αυτές διαθέτουν get και set μεθόδους που λειτουργούν ως reads και writes σε volatile μεταβλητές καθώς και ατομικές μεθόδους όπως η compareAndSet και οι αριθμητικές πράξεις

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

AtomicCounter.java

```
class AtomicCounterExample {
```

```
  static AtomicCounter c = new AtomicCounter();
  private static class IncrementCounter implements Runnable {
    public void run() {
      for (int i = 0; i < 100; i++) {
        if (i % 2 == 0)
          c.increment();
        else
          c.decrement();
      }
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread[] threads = new Thread[100];
    for (int i = 0; i < 100; i++)
      threads[i] = new Thread(new IncrementCounter());

    System.out.println(c.value());

    for (int i = 0; i < 100; i++)
      threads[i].start();

    for (int i = 0; i < 100; i++)
      threads[i].join();

    System.out.println(c.value());
  }
}
```

AtomicCounterExample.java

```
javac AtomicCounterExample.java
java AtomicCounterExample
```

```
0
0
```

ThreadLocalRandom

Το πακέτο java.util.concurrent περιέχει την κλάση ThreadLocalRandom για τη λήψη τυχαίων τιμών από πολλαπλά threads. Η χρήση της ThreadLocalrandom αντί για τη math.random() οδηγεί σε καλύτερη απόδοση.

Χρήσιμοι σύνδεσμοι

Java concurrent animated (https://sourceforge.net/projects/javaconcurrenta/)